ANALYSIS OF EXECUTABLE PROGRAM CODE USING COMPILER-GENERATED FUNCTION ENTRY POINTS AND ENDPOINTS WITH OTHER SOURCES OF FUNCTION ENTRY POINTS AND ENDPOINTS

Inventor(s)

Robert Hundt
870 E El Camino Real, #411
Sunnyvale, CA 94087

Vinodha Ramasamy
1257 Bracebridge Court
Campbell, CA 95008

Jose German Rivera
880 E Freemont Ave. #336
Sunnyvale, CA 94087

Umesh Krishnaswamy
940 Inverness Way
Sunnyvale, CA 94087

Assignee

Hewlett Packard Company

ANALYSIS OF EXECUTABLE PROGRAM CODE USING COMPILER-GENERATED FUNCTION ENTRY POINTS AND ENDPOINTS WITH OTHER SOURCES OF FUNCTION ENTRY POINTS AND ENDPOINTS

FIELD OF THE INVENTION

The present invention generally relates to analysis of executable program code, and more particularly to finding entry points and endpoints of functions in support of program analysis.

BACKGROUND

Analysis of binary executable programs is performed to analyze program performance, verify correctness, and test correct runtime operation, for example. Some analyses are performed prior to runtime (static analysis), while other analyses are performed during runtime (dynamic analysis). For both static and dynamic analysis, however, the analysis may be performed at the function level.

The term, "function", refers to named sections of code that are callable in the source program and encompasses routines, procedures, methods and other similar constructs known to those skilled in the art. The functions in the source code are compiled into segments of executable code. For convenience, the segments of executable code that correspond to the functions in the source code are also referred to as "functions".

A function is a set of instructions beginning at an entry point and ending at an endpoint. The entry point is the address at which execution of the function begins as the target of a branch instruction. The endpoint is the instruction of the function from which control is returned to the point in the program at which the function was initiated. For functions having multiple entry points and/or multiple endpoints, the first entry point and the last endpoint define a function.

The function entry points and endpoints of a program have in the past been obtained from symbol tables that are associated with the executable program code and from debug information that is present with the file having the executable program code (an "executable"). The debug information includes, for example, types of data entities, names of data entities, and the relationship between source and binary code. A symbol table contains a mapping of symbolic names of functions to entry points of the functions. However, since the generation of debug information is selectable at the compilation stage, some executables do not have debug information. Similarly, the presence of symbol tables is often optional, since in most environments a symbol table is unnecessary to run an executable. Thus, effective analysis of an executable presently depends on whether there is sufficient debug information or a complete symbol table.

A system and method that address the aforementioned problems, as well as other related problems, are therefore desirable.

SUMMARY OF THE INVENTION

The present invention provides in various embodiments a method and apparatus for analysis of executable program code. The executable program includes segments of code that correspond to callable functions in the source code from which the executable code was generated. Compiler-generated checkpoint descriptors are included in the executable and include pairs of entry points and endpoints. Each pair of entry points and endpoints is associated with a callable function in the source code. The pairs of entry points and endpoints are read from the executable program code and used to generate analysis data for the associated functions. In other embodiments, pairs of entry points and endpoints are additionally assembled from dynamic load modules and symbol tables.

It will be appreciated that various other embodiments are set forth in the Detailed Description and Claims which follow.

BRIEF DESCRIPTION OF THE DRAWINGS

Various aspects and advantages of the invention will become apparent upon review of the following detailed description and upon reference to the drawings in which:

FIG. 1 is a block diagram that illustrates an example data structure used in the management of checkpoints;

FIG. 2 is a flowchart of a process for finding function entry points and endpoints in executable program code in accordance with one embodiment of the invention; and

FIG. 3 is a flowchart of a process for analysis of a runtime detected function call.

DETAILED DESCRIPTION

In various embodiments, the present invention uses compiler-generated checkpoints to identify function entry points and endpoints in executable program code. The function entry points and end-points are then used to support analysis of the executable program code. Compiler-generated checkpointing is described in the patent/application entitled, "COMPILER-BASED CHECKPOINTING FOR SUPPORT OF ERROR RECOVERY", by Thompson et al., filed on October 31, 2000, and having patent/application number 09/702,590, the contents of which are incorporated herein by reference.

The process by which the compiler generates checkpoints generally entails identifying the points of program execution at which checkpoints are appropriate, and generating object code to implement the checkpoints. The checkpoints are identified at function boundaries. The executable program code is generated, along with executable checkpoint code. Generally, the executable checkpoint code saves the state of data objects to be recovered in the event of a hardware failure or a software exception, for example a C++ exception. A data structure is created to manage the data saved at each checkpoint in relation to the checkpoints in the program code. Since the checkpoints save the state of possibly different sets of data objects ("checkpoint data set"), each checkpoint has associated storage for state of the associated checkpoint data set.

FIG. 1 is a block diagram that illustrates an example data structure used in the management of checkpoints. Checkpoints in executable program code 102 delineate the constituent functions. In the example code, functions 104, 106, 108, and 110 have the associated checkpoints 112, 114, 116, and 118. The labels at the checkpoints represent the checkpoint code (or branch) that is executed to save checkpoint data.

Checkpoint data sets 122 includes a checkpoint data set for each of the checkpoints. For example, checkpoint 112 is associated with data set 124, and checkpoint 114 is associated with checkpoint data set 126.

Each checkpoint data set includes storage for the state of the data objects that are recorded at the associated checkpoint. It will be appreciated that for recovery purposes each of checkpoint data sets 122 may also include references (not shown) to the data objects that are associated with the state data.

For each function in the source code, the compiler generates an associated checkpoint descriptor. The term, "function", refers to named sections of code that are callable in the source program and encompasses routines, procedures, methods and other similar constructs known to those skilled in the art. The functions in the source code are compiled into segments of executable code. For convenience, the segments of executable code that correspond to the source code functions are labeled with "function" in FIG. 1.

As described in the referenced patent/application, the checkpoints can be used for recovery in the event of program failure. Relative to the present invention, the checkpoint descriptors 132 and 134 that are generated by the compiler are used in identifying functions and then generating analysis data for the functions. For example, checkpoint descriptor 132 is associated with function 104, and checkpoint descriptor 134 is associated with function 110. The checkpoint descriptors are stored in the executable program code file, which is also referred to as the "executable" or ELF (executable load file).

Each checkpoint descriptor includes an entry point and an endpoint. The entry point is the starting address of the associated function, and the endpoint is the address at which the function exits execution.

FIG. 2 is a flowchart of a process for finding function entry points and endpoints in executable program code in accordance with one embodiment of the invention. The process generally entails generating a list of pairs of entry points and endpoints of the functions in the executable. This list is referred to herein as the list of function entry-end points. The entry points of the functions are gathered from various sources, including for example, checkpoint descriptors generated in compilation of the code, a symbol table, and any dynamic load modules associated with the executable.

At step 302, the process searches for checkpoints in the executable program code 102 by looking for checkpoint descriptors (e.g., 132 and 134 of FIG. 1). If the executable does not contain any checkpoint descriptors, decision step 304 directs the process to step 306, where the starting address of the executable is added to the list of function entry-end points. The endpoint of the executable is the end of the binary segment. If checkpoint descriptors are found in the executable, at step 308 the entry points and associated end-points are read from the checkpoint descriptors and stored in the list of entry-end points. The process then continues at decision step 310.

Decision step 312 tests whether a symbol table is present for the executable. In many environments, a symbol table is optionally created during compilation of program source code at the discretion of the user. The symbol table includes a mapping of the symbolic names of functions in the executable to the function entry-points. If the symbol table is present, the process is directed to step 312, and the entry points from the symbol table are added to the list of function entry-end points. Otherwise, the process is directed to decision step 314.

Decision step 314 tests whether there are any dynamic load modules present with the executable. If so the process is directed to step 315. At step 315, the entry points of the initializer and de-initializer functions in the dynamic load modules are added to the list of entry-end points. The ending addresses of the binary segments are used as the endpoints of the initializer and de-initializer functions . At step 316, the production lookup tables (PLTs) associated with the dynamic load modules are checked for any additional functions. Any functions that are exported from a dynamic load module are identified in the associated PLT.

After step 316, or if there are no dynamic load modules loaded, the process is directed to step 318. At step 318 endpoints are identified for the entry points in the list. It will be appreciated that thus far, the only source of endpoints is checkpoint descriptors in the executable. If there are no checkpoint descriptors, no endpoints will be present in the list of entry-endpoints. Endpoints are identified and associated with those entry points in the list that do not have endpoints. The endpoints are determined using the function entry points in the list. For example, for an entry point p_1 of a function f_1 that precedes a function f_2 in the executable, where f_2 has an entry point p_2, the endpoint of f_1 can be determined by a known offset back from the entry point f_2 of p_2. For entry points for which endpoints cannot be identified, the end of the code segment is used for the respective endpoints.

At this juncture, the static detection of function entry points and endpoints is complete and static analysis of the executable can be performed. Further function detection can be performed at runtime. At step 320, the program is executed and function calls are detected. Function calls are characterized by execution of a branch instruction, for example, and the target address of the branch is the function entry point. Entry points detected at runtime are added to the list of entry-end points. Various runtime characteristics of the functions are recorded in accordance with the user-selected analysis.

The list of function entry points and endpoints supports various forms of binary analysis. Some example analyses supported by the present invention are described in the following paragraphs.

Disassembly of an executable can use the function entry points and endpoints to translate calls to addresses into calls to functions and output function calls, which are descriptive, instead of jumps addresses. In addition, individual functions can be disassembled without having to disassemble the entire executable.

The identification of function entry points and endpoints supports completeness and containment analysis. For example, it is possible to verify the existence of a given function in a load module.

A function matching technique can be employed with reference to the function entry points, endpoints, and executable code to detect copyright infringement. Analysis involving static instrumentation is also supported by the present invention.

Reverse engineering tools can also benefit from identification of function entry points and endpoints. For example, these tools attempt to generate a high-level language representation of an executable, which generally use constructs such as functions.

FIG. 3 is a flowchart of a process for analysis of a runtime detected function call. As described previously, a function call can be detected by execution of a branch instruction, for example. Upon detection, decision step 352 tests whether the called function is a stub function. A stub function is a segment of code that is target of a branch instruction and that does not return control to the point following the branch instruction after execution of the stub code segment. Stub functions can be detected using the symbol table, pattern matching, or searching for a "return" instruction in the code segment. For a stub function, the entry point is not saved and no analysis is performed.

Decision step 352 directs the process to decision step 354 for non-stub function calls. Decision step 354 tests whether the function entry point is already in the list of entry-endpoints. If not, the entry point is added to the list and a corresponding endpoint is generated at step 356. The endpoint is also added to the list. At step 358, selected execution characteristics of the function are recorded. It will be appreciated that the particular characteristics recorded depend on the user-selected analysis. This process is repeated for each detected function call.

Other aspects and embodiments of the present invention, in addition to those described above, will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein. It is intended that the specification and illustrated embodiments be considered as examples only, with a true scope and spirit of the invention being indicated by the following claims.